

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Spring Semester, 2006

Quiz II Solutions

These are example solutions to Quiz 2. In some cases, alternative answers were acceptable.

Part 1: (23 points)

For computations that do not involve mutation, it can sometimes be efficient to remember previous computations of a procedure, for example, storing the value obtained by applying a procedure to a particular argument. This technique, called **memoization**, allows us to avoid redoing the same computation at subsequent stages.

Here is some code for memoizing a procedure of one numeric argument, together with an example use:

```
(define (memoize proc)
  (let ((hist '()))
    (lambda (arg)
      (let ((prev (its-in arg hist)))
        (if prev
            prev
            (let ((new (proc arg)))
              (set! hist (cons (list arg new) hist))
              new))))))

(define (its-in arg hist)
  (cond ((null? hist) #f)
        ((equal? arg (caar hist))
         (cadar hist))
        (else (its-in arg (cdr hist)))))

(define my-sq (memoize (lambda (x) (* x x))))

(my-sq 5)
```

Attached at the end of the exam is a partially completed environment diagram corresponding to this set of evaluations. Complete that diagram, and then use the labels on the parts of the environment diagram to answer the following questions.

Question 1: For each of the following frames, indicate the enclosing environment, choosing one of **GE**, **E1**, **E2**, **E3**, **E4**, **E5**, **E6**, **E7** or **none** or **not shown**.

Frame:	Enclosing Environment
GE	none
E1	GE
E2	E4
E3	E6
E4	E3
E5	GE
E6	E1
E7	GE

Question 2: For each of the following procedure objects, indicate the enclosing environment, choosing one of **GE**, **E1**, **E2**, **E3**, **E4**, **E5**, **E6**, **E7** or **none** or **not shown**.

Procedure:	Enclosing Environment
P1	E6
P2	GE
P3	GE
P4	GE

Question 3: For each of the following variable names, indicate the value to which it is bound at the end of the evaluation of the expressions, choosing one of **GE**, **E1**, **E2**, **E3**, **E4**, **E5**, **E6**, **E7** or **P1**, **P2**, **P3**, **P4** or a symbol, a number, a boolean value, a list or the empty list.

Variable:	Environment	Value
memoize	GE	P4
its-in	GE	P2
my-sq	GE	P1
proc	E1	P3
new	E2	25
arg	E3	5
prev	E4	#f
arg	E5	5
hist	E5	'()
hist	E6	((5 25))
x	E7	5

Part 2: (18 points)

Consider the following two procedures for processing trees:

```
(define (tree-map proc tree)
  (cond ((leaf? tree) (proc tree))
        (else (map (lambda (x) (tree-map proc x)) tree))))

(define (leaf? x) (not (list? x)))

(define (tree-fold leaf-op combine init tree)
  (if (leaf? tree)
      (leaf-op tree)
      (fold-right combine init
                   (map (lambda (x) (tree-fold leaf-op combine init x))
                        tree))))

(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst)) (map proc (cdr lst)))))

(define (fold-right proc init lst)
  (if (null? lst)
      init
      (proc (car lst) (fold-right proc init (cdr lst)))))
```

As an example, here is a test tree:

```
(define test '(1 (2 (3 (4) 5) 6) 7))
```

Question 4: Write an expression using `tree-map` that will double the values of all the leaves of a tree (you may assume that the leaves only hold numerical values), e.g.,

```
>(tree-map ... test)
(2 (4 (6 (8) 10) 12) 14)

(tree-map (lambda (x) (* x 2)) test)
```

Question 5: Write an expression using `tree-map` that makes a copy of a tree, e.g.,

```
>(tree-map ... test)
(1 (2 (3 (4) 5) 6) 7)
```

```
(tree-map (lambda (x) x) test)
```

Question 6: Write an expression using `tree-fold` that makes a copy of a tree, e.g.,

```
>(tree-fold ... ... test)
(1 (2 (3 (4) 5) 6) 7)
```

```
(tree-fold (lambda (x) x) cons '() test)
```

Question 7: Write an expression using `tree-fold` that will flatten a tree, e.g.,

```
>(tree-fold ... ... test)
(1 2 3 4 5 6 7)
```

```
(tree-fold list append '() test)
```

Question 8: Write an expression using `tree-fold` that counts the number of leaves in a tree, e.g.,

```
>(tree-fold ... ... test)
7
```

```
(tree-fold (lambda (x) 1) + 0 test)
```

Question 9: Write an expression using `tree-fold` that returns `#t` if all of the leaves of the tree are odd (you may assume that `odd?` is defined, and that all the values are numbers), e.g.,

```
>(tree-fold ... ... test)
#f
```

```
(tree-fold (lambda (x) (if (number? x) (odd? x)) and #t test)
```

Part 3: (20 points)

Earlier in the term, we saw a set of procedures for processing lists, in particular, `map`, `filter`, `append`, and `reverse`. Now that we have introduced mutation, it is possible to create versions of these procedures that alter the list structure of their argument, rather than create a new version of the list structure. For each of the questions below, provide the code fragments necessary to complete these mutating versions of list procedures.

Question 10:

The procedure `map!` should mutate its list of arguments, replacing each value with the result of applying the supplied procedure. Here is a template, with the need portions completed in uppercase:

```
(define (map! f lst)
  (define (lter l)
    (cond ((null? l) lst)
          (else (SET-CAR! L (F (CAR L)))
                (ITER (CDR L)))))
  (iter lst))
```

Question 11:

The procedure `append!` should create a single list from two lists, by mutation. It's behavior is shown below:

```
>(define x (list 1 2 3))
>(define y (list 4 5 6))

>(append! '() x)
(1 2 3)

>(append! x y)
(1 2 3 4 5 6)

>x
(1 2 3 4 5 6)

>y
(4 5 6)
```

Here is the completed procedure with the missing portions in upper case.

```
(define (append! a b)
  (define (iter first-lst second-lst)
    (IF (NULL? (CDR FIRST-LST))
        (SET-CDR! FIRST-LST SECOND-LST)
        (ITER (CDR FIRST-LST) SECOND-LST)))
  (cond ((null? a) b)
        (else (iter a b)
              a)))
```

Question 12:

The procedure `reverse!` should reverse the elements of its argument “in place”, that is by mutating the existing list structure, rather than creating a new copy. It should do this by working down the list, keeping track of the last `cons` cell and the current `cons` cell, and mutating the structure so that the current cell points to the last one (i.e. reversing the order) while being careful to keep track of the rest of the list. When you are done, the original list may be destroyed (see example below):

```
>(define x (list 1 2 3 4 5))
```

```
>(reverse! x)
(5 4 3 2 1)
```

```
>x
(1)
```

Here is the completed procedure

```
(define (reverse! lst)
  (define (iter last current)
    (if (null? current)
        last
        (LET ((NEXT (CDR CURRENT)))
            (SET-CDR! CURRENT LAST)
            (ITER CURRENT NEXT))))
  (iter '() lst))
```

Part 4: (20 points)

A **queue** is a data structure, in which elements can only be added at the end of the queue, and elements can only be removed from the front of the queue (kind of like the line for LSC movies, assuming people behave courteously!). Traditionally, a queue is a tagged data structure where the elements are stored in a list, and the queue provides pointers to the front and end of the list. We are going to build a different implementation of a queue, in which the elements are represented by objects with state, including internal state variables pointing to the next element in the queue.

Here is part of the framework for such an implementation of a queue. We have filled in the missing parts in upper case.

```
(define (make-empty-queue)
  (list 'queue #f #f)) ; list of a tag, a pointer to the front of
                      ; the queue, and a pointer to the end of the queue

(define (queue? q)
  (tagged-list q 'queue))

(define (make-entry element)
  (let ((next #f))
    (lambda (msg)
      (cond ((eq? msg 'value) element)
            ((eq? msg 'next) next)
            ((eq? msg 'set-next!)
             (lambda (val) (set! next val)))))))

(define (insert-queue element q)
  ;; procedure to add entry to end of queue
  (let ((current-rear (caddr q))
        (new (make-entry element)))
    (cond ((not current-rear) ;;the queue is currently empty
           (SET-CAR! (CDR Q) NEW)
           (SET-CAR! (CDDR Q) NEW))
          (else
           ((CURRENT-REAR 'SET-NEXT!) NEW)
           (SET-CAR! (CDDR Q) NEW))))))

(define (delete-queue q)
  ;; procedure to remove entry from front of queue,
  ;; and return value of that entry
  (if (not (cadr q))
      (error "empty queue")
      (LET ((HOLD ((CADR Q) 'VALUE)))
        (SET-CAR! (CDR Q) ((CADR Q) 'NEXT))
        HOLD)))
```

Part 5: (19 points)

A stack is a data structure for a sequence of elements, with the property that elements can only be “pushed” onto the top of the stack, and “popped” off the top of the stack.

Here is a partial implementation of a stack, using a tagged data structure:

```
(define (make-empty-stack)
  (list 'stack '())) ;; a stack is a pair: a tag, and a list of elements

(define (tagged-list? element tag)
  (if (pair? element)
      (eq? (car element) tag)))

(define (stack? st)
  (tagged-list? st 'stack))

(define (push element stack)
  (if (stack? stack)
      (set-car! (cdr stack) (cons element (cdr stack)))))
```

Question 18: The procedure `pop` should take as an argument a stack, and should alter the stack to remove the first element, and return that value. If the stack is empty, it should return an error. Here is the completed procedure.

```
(define (pop stack)
  (if (stack? stack)
      (IF (PAIR? (CDR STACK))
          (LET ((RETURN (CAR (CDR STACK))))
              (SET-CDR! STACK (CDR (CDR STACK)))
              RETURN)
          (ERROR "SOME MESSAGE"))
      (error "not a stack")))
```

Question 19: We can create a different version of a stack, by requiring that the elements of the stack are ordered, based on some comparison predicate. For example:

```
> (define test (make-empty-stack))
> (ordered-push '(1 a) test (lambda (x y) (< (car x) (car y))))
> (ordered-push '(0 b) test (lambda (x y) (< (car x) (car y))))

> test
(stack (0 b) (1 a))

> (pop test)
(0 b)
```

Here is a template for the procedure `ordered-push`:

```
(define (ordered-push element stack less?)
  (if (stack? stack)
      (set-car! (cdr stack) (insert-ordered less? element (cdr stack)))))
```

Write the procedure `insert-ordered`:

```
(define (insert-ordered less? elt lst)
  (cond ((null? lst) (list elt))
        ((less? elt (car lst))
         (cons elt lst))
        (else (cons (car lst) (insert-ordered less? elt (cdr lst))))))
```